Inf2a: Lab 1
Introduction to Python

This short overview of Python. A more comprehensive tutorial for Python 2.6 can be found online at:

`http://docs.python.org/2.6/tutorial/`

That tutorial offered by Python.org contains more on control statements, functions and their definitions, data structures, modules, classes and the Python library.

1 Interpreter

In this session we will launch the Python interpreter and directly execute commands in the terminal window. Python is an interpreted language, so one of its great advantages is the ability to launch an interpreter in this way. This can save a lot of time when you are developing software: when you are not sure about something, you can often test it directly from the interpreter, without the overhead of having a small program to launch from command line. The interpreter is launched by typing the following command in the shell:

```
python
```

This launches the default version of Python on DICE (version 2.6). Once python has been launched, lines previously inserted in the interpreter shell can be recalled by pressing the ↑ cursor button. (The ← and → cursor buttons also work as expected to move backwards and forwards through the characters in the current line.) To exit the interpreter, type `<CTRL>+D`

In Python, **formatting has meaning** - this means the spacing is very important. Groups of statements are indented under a header. Blocks of code are nested by increasing the indentation. There are no end-of-line symbols (like the semicolon; in Java or C). Instead, newline marks the end of a line.
Below we will introduce you to the basic types.

⋆ In this lab, type the lines introduced by `>>>` and by `...` Cutting and pasting whole sections will disrupt the formatting and will not work.

1.1 Basic Types

1.1.1 Numbers

Python can be used as a simple calculator. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages.

⋆ Enter the following expressions in the Python interpreter you have just launched:

```
>>> 2+2
>>> 3*2
>>> 2 + 7.3
```

1.1.2 Strings

Strings can be enclosed in single or double quotes:

```
>>> 'hello'
>>> 'how\'s it going?'
>>> "how's it going?"
>>> 'This is "strange"'
```

If a string is enclosed in a pair of triple quotes, either `"""` or `'''`, it retains whatever formatting is found between them — for example:

```
>>> """
... This is a string that when
... printed maintains its format
... and its end of lines.
... """
>>> print """
... This is a string that when
... printed maintains its format
... and its end of lines.
... """
```

Strings can be concatenated using the `+` operator, and can be repeated with the operator `*`:

```
>>> 'Hello ' + ', ' + 'how are you'
>>> 'help' + '!'*5
```

The individual characters of a string can be accessed using indices. For example the first character has index 0. Substrings can be specified with slice notation, two indices separated by colon. When using slices, indices can be thought of as pointing **between** characters, instead of pointing to characters: the left edge of the first character is 0 and so on.

The following commands illustrate this with the help of a user-defined variable called `word`. Take care when you name your variables not to use reserved words. For a list of these, look here:
`http://docs.python.org/2/reference/lexical_analysis.html#keywords`.

```
>>> word="hello"
>>> word[0]
>>> word[2]
>>> word[0:2]
>>> word[2:5]
>>> word[:2]
>>> word[2:]
```

Negative indices start counting from the right end.

```
>>> word[-1]
>>> word[-2:]
```

Strings cannot be modified once they are created. Trying to modify a substring results in an error. However, it is easy to create a new string by concatenating substrings from other strings.

1.1.3 Lists

There are different types of compound structures in Python. The most versatile is the list.

```
>>> L = ['monday','tuesday','wednesday','thursday','friday']
>>> L
```

Like strings, list items can also be accessed using indices. Similarly, they can be sliced and concatenated:

```
>>> L[0]
>>> L[3]
>>> L[1:]
>>> L[:2]
>>> L[1:3]
>>> L + ['saturday','sunday']
```

One can verify the membership of an element to a list using the keyword `in`.

```
>>> 'wednesday' in L
>>> 'sunday' in L
```

Items can be added at the end of a list using the `append(item)` method of the `list` object.

```
>>> L3 = []
>>> L3.append(1)
>>> L3.append(2)
>>> L3
```

It is possible to measure the length of a list using the built-in function `len(list)`.

```
>>> len(L)
```

It is also possible to create nested lists:

```
>>> L1 = [1, 2, 3]
>>> L2 = ['one',L1,'two']
>>> L2
```

◦ `insert(i,x)` inserts the item x at position i.

```
>>> L2 = ['a','b','d','e']
>>> L2.insert(1,'c')
>>> L2
```

◦ `remove(x)` removes the first item in the list whose value is x.

```
>>> L2.remove('d')
>>> L2
```

◦ `pop()` returns (and removes) the last item in the list; likewise, `pop(i)` returns and removes the item in position `i`.

```
>>> L2.pop()
>>> L2
```

◦ `sort()` sorts the items of the list, in place.

```
>>> L2.sort()
>>> L2
```

◦ `reverse()` reverses the elements of the list, in place.

```
>>> L2.reverse()
>>> L2
```

◦ `del` can be used to remove items from a list using the index. It can also be used to remove slices from a list.

```
>>> del L2[1:3]
>>> L2
```

You can iterate over a list retrieving the index and the value at the same time using the `enumerate(list)` function.

```
>>> for i, v in enumerate(['a','b','c','d']):
...    print i, v
...
```

1.1.4 Boolean Types

The Boolean type is named bool; it takes any Python value and converts it to True or False.

```
>>> bool(1)
>>> bool(0)
>>> bool([1])
>>> bool([])
```

1.1.5 Tuples

A *tuple* is composed by a number of values separated by commas, and enclosed by parentheses.

```
>>> T = (1,2,'three')
>>> T
>>> T[2]
```

Tuples can be nested.

```
>>> T1 = (1,2,(3,5))
>>> T1
```

Tuples, like strings, are immutable and cannot be changed once created. If you try, you will get an error message.

```
>>> s = "hello"
>>> s[1] = "u"
>>> T1[2] = 3
```

1.1.6 Dictionaries

Dictionaries are very useful structures. They are indexed by **keys**, unlike lists which are indexed by indices. The keys can be any immutable objects (numbers, strings, tuples - you cannot use lists, which are mutable).

A dictionary can be seen an unordered set of *key:value* pairs, with the constraint that keys need to be unique. The main operations performed on dictionaries are storing and retrieving values by their keys.

```
>>> num = {'one':1, 'two':2, 'three':3, 'four':4}
>>> num['three']
>>> num
```

You can easily add a new item to the dictionary:

```
>>> num['five'] = 5
>>> num
```

You can delete one item from the dictionary using the built in function `del(item)`:

```
>>> del(num['three'])
>>> num
```

To list all the keys from a dictionary, you use the method `keys()`. To check if a key belongs to the dictionary you use the method `has_key()`:

```
>>> num.has_key('one')
>>> num.keys()
```

You can iterate over a dictionary, retrieving the keys and their corresponding values using the method `iteritems()`:

```
>>> for k, v in num.iteritems():
...     print k,v
...
```

1.2 Modules

Programs can become long, and it is a good approach to divide them into more manageable units. In Python, programs can be divided into **modules**. The modules can be imported directly into the interpreter or into other programs. Python has a wide library of predefined functions and classes that can be imported and used.

```
>>> import math
```

To call a function imported in a module, it must be prefixed with the module name, to avoid name conflicts. (N.B. The function `math.ceil(x)` below returns as a float, the smallest integer greater than or equal to x.)

```
>>> math.pi
>>> math.ceil(2.35)
```

We can also import only the functions we need. If we do this, it is not necessary to prefix them with the module name.

```
>>> from math import ceil
>>> ceil(2.35)
```

`dir(modulename)` will return a list of names defined in the module. If you modify a module, you need to reload it using `reload(modulename)`.

1.2.1 Pattern Matching

The `re` module provides tools for regular expressions.

```
>>> import re
```

◦ `match(pattern, string)` if zero or more characters at the beginning of `string` match the regular expression `pattern`, it returns a corresponding `MatchObject` instance. It returns `None` if the string does not match the pattern. For a guide to writing regular expressions in machine syntax, see the slides for Informatics 2A Lecture 6.

```
>>> re.match("(aa|bb)+", "aabbaa")
>>> re.match("(aa|bb)+", "abba")
```

◦ `findall(pattern, string)` returns a list of all non-overlapping matches of `pattern` in `string`.

```
>>> re.findall("[a-z]*th[a-z]*", "I think this is the right one")
```

◦ `sub(pattern, repl, string)` returns the string obtained by replacing the leftmost non-overlapping matches of `pattern` in `string` by the replacement `repl`.

```
>>> re.sub("[a-z]*th[a-z]*", "TH-word", "I think this is the right one" )
```

⋆ End your interpreter session using `<CTRL>+D`.

2 Text Editors

In the rest of this lab we will use the types explained above. You may find it useful to refer back.

From here on we will use Python scripts rather than using the interpreter directly. Launch your preferred editor (emacs, vim, gedit,...) in the background (using &). If you need further advice on using an editor try an emacs tutorial such as the one within emacs itself: type "C-h t" (control-h, t) within emacs.

Remember: in Python, formatting has meaning - this means the spacing is very important. Groups of statements are indented under a header. Blocks of code are nested by increasing the indentation. There are no end-of-line symbols (like the semicolon; in Java or C). Instead, newline marks the end of a line.

⋆ Before lauching Python, open a terminal window and create a new directory called "`MyPython`":

```
mkdir MyPython
```

⋆ Go to this directory:

```
cd MyPython
```

2.1 Control Structures

Python offers the usual control flow statements, as do other languages like Java or C. The `for` loop is more powerful than in most of the other languages.

At this point you might want to write the commands as a program in an editor (such as emacs, vim or gedit), so that you can easily modify the lines in case of errors. You can then execute the program from the shell.

For example, to open the emacs editor, type the following command from the shell:

```
emacs program_name.py &
```

Then write your program in the editor. To save the program, press `<CTRL>+X` and then `<CTRL>+C`. This will prompt you to save the file. Press 'y'. This will take you back to shell.

To launch your program, type the following command from the shell:

```
python program_name.py
```

Now we will consider the python control structures `if`, `while` and `for`.

2.1.1 If Statement

⋆ Using an editor, create a file `binary.py` containing the following lines:

```
x=raw_input("Enter a binary sequence : ")
if (x[0] == '0'):
    print "Starts with 0"
elif (x[0] == '1'):
    print "Starts with 1"
else:
    print "Error: Not a binary number!"
```

⋆ After saving the file `<CTRL>+X` and then `<CTRL>+S`, launch it from the shell.

```
python binary.py
```

⋆ When asked, insert a binary sequence (such as 00101 or 10010)

It is possible to create more complex conditions using the keyword and for conjunction and the keyword or for disjunction – e.g.

```
if (x[0] == '0' or x[0] == '1'):
    print "Starts with " + x[0]
else:
    print "Error: Not a binary number!"
```

2.1.2 While

⋆ Using the editor create a file `downtoone.py`. Type the following lines.

```
x = 10
while (x > 0):
    print x
    x = x - 1
```

⋆ Save the file and then launch the program from the shell.

2.1.3 For Loop

The `for` statement iterates over the items of any sequence (such as strings or lists), using the keyword `in` discussed earlier.

⋆ Using an editor, create a file `printlist.py` . Type the following lines.

```
L = ['monday','tuesday','wednesday','thursday','friday']
for x in L:
    print x
```

⋆ Save the file and then launch the program from the shell.

To iterate over a sequence of numbers, the built-in function `range()` can be used to automatically generate the sequence:

⋆ Re-enter the python interpreter and type in the following lines.

```
>>> range(5)
>>> range(5,10)
```

⋆ Using an editor, create a file `printnumbers.py`. Type the following lines.

```
for x in range(10):
    print x
```

⋆ Save the file and then launch the program from the shell.
You can write an `if` statement nested inside a `for` statement as follows, which prints the even numbers less than 10. Remember that nesting is indicated by increased indentation.

⋆ Using an editor, create a file `printevennumbers.py` and enter the following lines, where `%` is the "modulo" operator.

```
for x in range(10):
    if (x%2 == 0):
        print x
```

⋆ Save the file and then launch the program from the shell.

2.2 Functions

A function is introduced by the keyword `def`, followed by the function name and by its list of parameters in parentheses. The statements that form the function body start on the next line, and are indented. The first line can optionally be a string that describes the function. This string can be used by automatic generators of documentation.

You can either type the code in the interpreter or alternatively type it as a program and execute it from shell.

```
>>> def square(value):
...  """Returns the square value of the value"""
...  return value*value
...
>>> square(4)
```

3 Exercises

EXERCISE 1

Create a regular expression that checks if a string starts with 3 binary digits (and test it: `010asda` must be recognised, while `1aa` must be rejected)

Using a regular expression, write a python statement that finds all the words that end with "`ly`" in strings (and test it, for example using the sentence `"it is likely to happen rarely"`)

Using a regular expression, write a python statement that replaces all the words that start with "`wh`" by "`WH-word`" (and test it, for example in the sentence `"who should do what?"`)

EXERCISE 2

Write a program that, for every element in the list
`['how', 'why', 'however', 'where', 'never']`
prints:

- a star symbol '*'
- the first two letters from the element (its 2-character prefix)
- the whole element

i.e. producing something like:

```
* ho how
* wh why
...
```

EXERCISE 3

Modify the previous program to print a star ('*') in front of the elements that start with the prefix "wh" and a hyphen '-' in front of the others, producing something like:

```
- ho how
* wh why
...
```

EXERCISE 4

Write a function `checkPrefix(list, prefix)` in the python interpreter. This function should wrap the loop created in Exercise 2. When the function is called, it should print the contents of `list`, adding a star in front of those elements that start with the two-character prefix in `prefix`. Check what happens when `checkPrefix` is given an empty `list`.

EXERCISE 5

Using an editor, create a new file and type in the function `checkPrefix(list,prefix)`. Save the file as "`checker.py`" in your "`MyPython`" directory. Then import your module in the python interpreter and call the function.